



- (51) International Patent Classification:
G06F 9/30 (2006.01) *G06F 9/48* (2006.01)
- (21) International Application Number:
PCT/US2017/015052
- (22) International Filing Date:
26 January 2017 (26.01.2017)
- (25) Filing Language: English
- (26) Publication Language: English
- (30) Priority Data:
15/052,765 24 February 2016 (24.02.2016) US
- (71) Applicant: INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95054 (US).
- (72) Inventors: KIDD, Taylor, W.; 1243 Valley Quail Circle, San Jose, CA 95120 (US). WALSH, Matt, S.; 2252 Howard Avenue, San Carlos, CA 94070 (US).
- (74) Agent: PARVIN, S., Kameron; Nicholson De Vos Webster & Elliott, LLP, 217 High Street, Palo Alto, CA 94301 (US).
- (81) Designated States (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.
- (84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU,

[Continued on next page]

(54) Title: SYSTEM AND METHOD FOR CONTEXTUAL VECTORIZATION OF INSTRUCTIONS AT RUNTIME

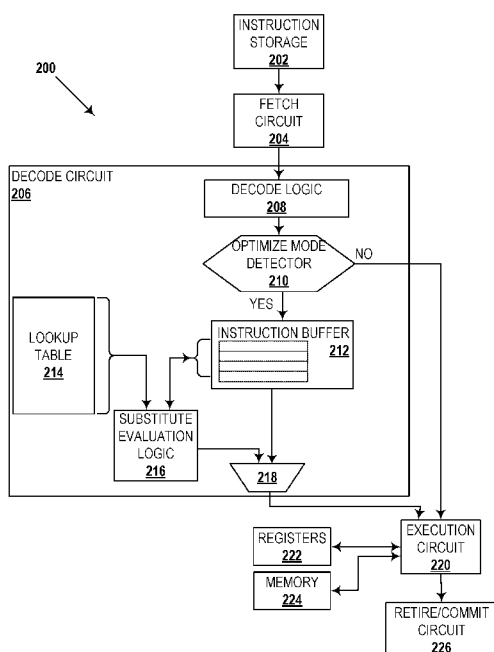


FIG. 2

(57) Abstract: Methods and apparatuses relating to processors that contextually optimize instructions at runtime are disclosed. In one embodiment, a processor includes a fetch circuit to fetch an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction. The disclosed processor further includes a decode circuit to decode the instruction, to detect the start-of-sequence instruction and the end-of-sequence instruction, to buffer the sequence of sub-optimal instructions there between, to access a lookup table to identify one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and to select either the decoded instruction or the sequence of one or more optimized instructions to dispatch to an execution circuit.



LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK,
SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,
GW, KM, ML, MR, NE, SN, TD, TG).

— *before the expiration of the time limit for amending the
claims and to be republished in the event of receipt of
amendments (Rule 48.2(h))*

Published:

— *with international search report (Art. 21(3))*

System and Method for Contextual Vectorization of Instructions at Runtime

Technical Field

[0001] Embodiments described herein generally relate to processors. In particular, embodiments described generally relate to processors configured to contextually optimize instructions at runtime.

Background Information

[0002] Parallel processing is generally faster than scalar execution of one data point at a time. Single Instruction Multiple Data (SIMD) computers with multiple processing elements that perform the same operation on multiple data points achieve performance gains by taking advantage of parallelism and simultaneous use of multiple parallel execution cores.

[0003] SIMD processors can take advantage of parallelism in performing mathematical operations and in moving data. SIMD processors can load or store multiple data items simultaneously, resulting in a performance gain compared to the slower, scalar processors that load or store one datum at a time. When executing computer programs on a processor with parallel resources, utilizing SIMD instructions offers better performance than utilizing scalar instructions.

[0004] Programming using a SIMD Instruction Set Architecture (ISA), however, can be challenging. SIMD ISA's, for example, are generally processor-specific. Programs that use SIMD instructions may need to be rewritten and customized to suit a new processor generation. The work required to adapt scalar instructions to a new instruction set architecture, including rewriting the code, documenting the code; enabling compilers to emit the code, training users to use the code, and to debug and collect traces of code execution may need to be partly or wholly repeated for reuse with each new generation of instruction set architecture (e.g. MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX 3.1, and AVX 3.2). What is needed, therefore, is a way to allow programmers to take advantage of

SIMD Instruction Set Architectures while avoiding the challenges inherent in conventional solutions.

[0005] Furthermore, conventional solutions are limited because they optimize the code statically, ahead of time, rather than dynamically, during execution. Compilers attempt to optimize execution of certain code sequences, but they operate in a static environment, without knowledge of the state of the machine or of the registers. Even SIMD code that was conventionally coded by hand is not able to optimize the code according to the run-time state of the machine and of the registers. What is needed, therefore, is a way to optimize instructions at run-time, with knowledge of the state of the registers and their contents.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] **Figure 1** is a block flow diagram illustrating a procedure for a processor to contextually optimize instructions at runtime according to one embodiment.

[0007] **Figure 2** is a block diagram illustrating processing components used by a processor to contextually optimize instructions at runtime according to one embodiment.

[0008] **Figure 3** illustrates an instruction and its various fields according to one embodiment.

[0009] **Figure 4** illustrates an exemplary register allotment by a processor optimizing Vectorbeam code of Table 2 according to an embodiment.

[0010] **Figure 5** illustrates an exemplary allotment of multiple computational resources used to process the allotted registers of **Figure 4** in parallel according to one embodiment.

[0011] **Figure 6** illustrates a portion of a lookup table listing vector instruction substitutes for scalar instructions according to one embodiment.

[0012] **Figure 7** illustrates a vector processor register file according to one embodiment.

DETAILED DESCRIPTION

[0013] In the following description, numerous specific details are set forth. However, it is understood that embodiments of the disclosure may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description.

[0014] References in the specification to “one embodiment,” “an embodiment,” “an example embodiment,” etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment need not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

[0015] A (e.g., hardware) processor, or set of processors, executes instructions from an instruction set, e.g., the instruction set architecture (ISA). The instruction set is the part of the computer architecture related to programming, and generally includes the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). It should be noted that the term instruction herein may refer to a macro-instruction, e.g., an instruction that is provided to the processor for execution. A processor (e.g., having one or more cores to decode and/or execute instructions) may operate on data, for example, in performing arithmetic, logic, data movement, or other functions.

Contextual Optimization at Runtime

[0016] According to the disclosure, a processor contextually optimizes instructions at runtime. Specifically, the processor optimizes a sequence of sub-optimal instructions dynamically at runtime. Sub-optimal instructions as used herein refer to instructions that make less-than-full use of the available resources of a processor, or which can be

optimized to take advantage of a processor's instruction set architecture. In one embodiment, sequences of sub-optimal instructions are stored in instruction storage, and are surrounded by start- and end-delimiting instructions. Generally, for purpose of disclosure, the sequence of sub-optimal code surrounded by start-of-sequence and end-of-sequence instructions is referred to herein as Vectorbeam code. The claims herein are not limited by the name Vectorbeam. In alternate embodiments, the code can be referenced by different names. The Vectorbeam instructions are not limited in their choice of particular opcodes or mnemonics.

[0017] According to a disclosed embodiment, a processor is configured to detect a Vectorbeam code sequence of sub-optimal instructions, to buffer the sequence, and to access a lookup table to identify one or more instructions to substitute for and optimize one or more of the sub-optimal instructions of the Vectorbeam code sequence.

Delimiters: Start-of-Sequence and End-of-Sequence

[0018] In one embodiment, the start-of-sequence instruction and end-of-sequence instruction can be chosen to provide the processor with a hint on how to optimize the code sequence. In one embodiment, a processor can surmise that the sequence of sub-optimal instructions is meant to be an iterative loop because the sequence of sub-optimal instructions is preceded by a start-of-sequence delimiting instruction, "foreach," and is followed by an end-of-sequence delimiting instruction, "next." Similarly, in alternate embodiments, the start-of-sequence instruction may hint that the sequence of sub-optimal instructions is iterative by using delimiting instructions "foreach," or "do until," or "repeat until," or "do while." The end-of-sequence instruction may be similarly named so as to suggest its function. (e.g. "end," "exit," "stop," "continue," "return"). The start-of-sequence and end-of-sequence instructions may also have predetermined values that are not human readable, or are numeric, or have been selected automatically or randomly without human involvement.

Sequence of Sub-Optimal Instructions

[0019] A sequence of sub-optimal instructions, as used herein, refers to computer instructions that take less-than-full advantage of the parallel resources available in a processor on which those instructions are meant to be executed.

[0020] For example, the sequence of sub-optimal instructions may be an iterative, scalar loop, which, as written, uses fewer parallel resources than are available in the processor.

[0021] As a further example, the sequence of sub-optimal instructions may have been written to use 64-bit registers, but the processor on which those instructions are to be executed has 128-bit registers available.

[0022] As a further example, the sequence of sub-optimal instructions relate to multimedia operations, such as dimming the brightness of a screen of pixels. In such an example, scalar mathematical operations can be replaced with wide vector instructions.

Metadata

[0023] According to the disclosure, the start-of-sequence instruction may include operands that provide metadata to a processor that will implement the Vectorbeam code sequence. In one embodiment, the metadata gives the processor a hint as to how to optimize the code. In one embodiment, the sequence of sub-optimal instructions is preceded by a start-of-sequence instruction, “for each rax, 0, 64, 1” The metadata, “rax, 0, 64, and 1,” provide the processor with a hint that the sub-optimal code should be looped using register rax as a loop index, varying rax 64 times, starting at 0, with a stride of 1 for each iteration.

Description of Illustrative Embodiments

[0024] **Figure 1** is a block flow diagram illustrating a procedure for a processor to contextually optimize instructions at runtime according to one embodiment. Specifically, a processor that is configured to execute a procedure to contextually optimize instructions at runtime according to block flow diagram 100 fetches an instruction at 102, decodes the instruction at 104, and tests whether it is in optimize mode at 106. In one embodiment, the instruction to be fetched is stored in an instruction buffer. In alternate

embodiments, the instruction may be stored in an instruction register, a general register, a program stack, or a memory, including static and dynamic random access memories. A processor performing the procedure of flow diagram 100 decodes the fetched instruction (e.g., macro-instruction) at 104 to generate as an output one or more micro-operations, micro-code entry points, micro-instructions, other instructions, or other control signals.

[0025] According to the disclosure, a Vectorbeam code sequence consisting of a sequence of sub-optimal instructions is stored in the memory, preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction. In one embodiment, the start-of-sequence instruction may be coded in human-readable assembly code, and have mnemonics defined in a way so as to suggest how the sub-optimal instructions should be optimized. For example, the start-of-sequence instruction may be “for each,” or “do until,” or “repeat until,” or “do while,” to name a few examples. The end-of-sequence instruction may be similarly named so as to suggest its function. (e.g. “end,” “exit,” “stop,” “continue,” “return,” “abort,” “undo,” or “commit”). The start-of-sequence and end-of-sequence instructions may also have predetermined values that are not human readable, or are numeric, or have been selected automatically or randomly without human involvement.

[0026] The Vectorbeam code’s sub-optimal code sequences, in one embodiment are written to mimic scalar mnemonics and appear to have a scalar and/or serial flow of execution, making them easy to understand and to debug. Delimiting the sub-optimal code sequences with easy-to-understand delimiters may enhance the readability of the code, and may also suggest to the processor how to optimize the code at runtime.

[0027] If the processor determines at 106 that it is not running in optimize mode, it dispatches the decoded instruction to be executed at 114, and commits or retires the instruction at 116.

[0028] Entering the optimize mode in one embodiment occurs during run-time when the instruction decoded at 104 is a start-of-sequence instruction, as described above. In another embodiment, the processor enters the optimize mode at power-on, or after

reset. In another embodiment, the processor enters optimize mode implicitly after fetching a particular sequence of instructions at 102, decoding them at 104, and detecting one or more sub-optimal instructions, such as scalar instructions, or instructions from an older instruction set architecture. In an alternate embodiment, the processor enters optimize mode implicitly during run-time in response to a predefined run-time event.

[0029] Exiting the optimize mode in one embodiment occurs during run-time when the instruction decoded at 104 is an end-of-sequence instruction, as described above. In an alternate embodiment, the processor exits the optimize mode implicitly after occurrence of a condition, such as failing to identify a substitute instruction for a time. In an alternate embodiment, the processor remains in the optimize mode indefinitely.

[0030] If the processor determines at 106 that it is already in optimize mode, or if the processor detects a start-of-sequence instruction, it enqueues the decoded instruction at 107 into an instruction buffer. The instruction buffer in one embodiment is a series of general purpose registers. In other embodiments, the instruction buffer may utilize instruction registers, scalar registers, shift registers, a stack memory, a static or dynamic RAM memory, or a cache memory.

[0031] In operation and during run-time, the processor compares the instructions in the instruction buffer to a lookup table at 108 to determine whether suitable substitute instructions are available, including substitute instructions that would yield better performance. In evaluating at 108 whether suitable substitute instructions are available, the processor in one embodiment is guided by the metadata included with the start-of-sequence delimiter. In evaluating at 108 whether suitable substitute instructions are available, the processor in one embodiment is guided by the state and contents of the register file, described further below with respect to **Figure 7**.

[0032] Examples of the disclosed processor implementing Vectorbeam code and being guided by metadata are discussed below with reference to **Table 1** and **Table 2**. In one embodiment, the sub-optimal instructions of the Vectorbeam code comprise scalar instructions, and the substitute instructions comprise vector instructions or SIMD

instructions. In one embodiment, the sub-optimal instructions of the Vectorbeam code include instructions from an older generation processor's instruction set architecture, and the substitute instructions are selected from the instruction set of a more recent processor. In one embodiment, the sequence of sub-optimal instructions of the Vectorbeam code consists of an iterative loop of instructions, and the substitute instructions complete the operation with fewer loops, or with the use of vector instructions. In another embodiment, the sequence of sub-optimal instructions in the instruction buffer consists of a conditional loop, and the substitute instructions complete the operation with fewer loops, or with the use of vector instructions. If the processor determines at 108 that a substitute instruction that would yield better performance is available, it selects one or more substitute instructions at 110.

[0033] At 112, the processor selects the decoded instruction for execution if no substitute instructions were available at 108, or selects the one or more substitute instructions that were selected at 110. In one embodiment, the processor maintains a sequence of instructions in the instruction buffer, and considers multiple instructions when evaluating potential substitutes at 108. In another embodiment, the processor limits the number of instructions stored in the instruction buffer, and, if the number of instructions exceeds that limit, the processor removes one or more instructions from the instruction buffer and dispatches them to be executed at 114.

[0034] At 114, the processor executes the decoded instruction or the one or more substitute instructions. After execution, the processor retires or commits the instruction at 116, ensuring that execution results are written to or have been written to their destinations, and freeing up or releasing resources for later use.

[0035] **Figure 2** is a block diagram illustrating processing components used by a processor to contextually optimize Vectorbeam code instructions at runtime according to one embodiment. Specifically, block diagram 200 includes instruction storage 202, fetch circuit 204, decode circuit 206, execution circuit 220, and retire/commit circuit 226. Decode circuit 206 includes decode logic 208, optimize mode detector 210, instruction buffer 212, lookup table 214, substitute evaluation logic 216, and instruction selector 218.

Block diagram 200 further includes execution circuit 220, registers 222, memory 224, and retire/commit circuit 226.

[0036] In operation, the processor utilizes fetch circuit 204 during run-time to fetch an instruction from instruction storage 202. In one embodiment, instruction storage 202 is a register file. In alternate embodiments, instruction storage 202 may be an instruction buffer, an instruction register, a general register, a program stack, or a static or dynamic random access memory.

[0037] The processor passes the fetched instruction to decode circuit 206, which decodes the fetched instruction with decode logic 208. The instruction, including its opcode and optional operands, is described below with reference to **Figure 3**. The processor detects whether it is in optimize mode with optimize mode detector 210. If the processor is not in optimize mode, the processor dispatches the decoded instruction to execution circuit 220.

[0038] But, if optimize mode detector 210 detects that the processor is in optimize mode or sets the processor in optimize mode, the decoded instruction is queued in instruction buffer 212. As shown, instruction buffer 212 has four entries, but the number of entries is variable: it could be fewer than four, it could be greater than four, or it could be adjusted dynamically, as would be understood by those of skill. During run-time, the processor uses substitute evaluation logic 216 to evaluate the decoded instructions in instruction buffer 212.

[0039] Specifically, substitute evaluation logic 216 accesses lookup table 214 during run-time to determine whether any substitute instructions are available. The lookup table in one embodiment compares mnemonics of sub-optimal Vectorbeam instructions in the instruction buffer to a listing of substitute instructions. In one embodiment, the substitute instructions are vector instructions. In another embodiment, the substitute instructions are SIMD instructions. In another embodiment, the substitute instructions are from a newer instruction set architecture than the sub-optimal instructions. In evaluating substitutes, substitute evaluation logic 216 in one embodiment is guided by

metadata provided with a start-of-sequence instructions. In evaluating substitutes, substitute evaluation logic 216 in one embodiment evaluates the run-time state of the processor registers. For example, if multiple registers are used in determining memory addresses, and if those memory addresses fall in the same cache line, substitute evaluation logic 216 substitutes a vector memory access for scalar memory accesses associated with those memory addresses.

[0040] In one embodiment, if substitute evaluation logic 216 determines that a substitute instruction is available, the substitute instruction is passed to dispatch selector 218 to be dispatched for execution. In alternate embodiments, the substitute instruction is written into instruction buffer 212, adding to or replacing instructions therein. As shown, decode circuit 206 uses dispatch selector 218 to select instructions to dispatch either from instruction buffer 212 or from substitute evaluation logic 216.

[0041] Execution circuit 220 in one embodiment is a vector processor. In alternate embodiments, execution circuit 220 may include multiple cores, and parallel hardware. Execution circuit 220 in one embodiment utilizes registers 222 and memory 224 to store intermediate results and to otherwise support execution. After execution, retire/commit circuit 226 ensures that execution results are written to or have been written to their destinations, and frees up or releases resources for later use.

[0042] **Figure 3** illustrates an instruction and its various fields according to an embodiment. Specifically, instruction 300 includes opcode 302 and optional first, second, and third operand identifiers, 304, 306, and 308, respectively. Opcode 302 identifies the instruction and/or the operation to be performed, as well as the type of operands (e.g., “start-of-sequence” or “end-of-sequence”). First, second, and third operand identifiers, 304, 306, and 308, are optional. In one embodiment, opcode 302 is a “start-of-sequence” instruction and first, second, and third operand identifiers 304, 306, and 308 represent metadata that describe the sub-optimal instructions that follow, providing a hint or a suggestion to the processor as to how to optimize the sub-optimal instructions. In an alternate embodiment, opcode 302 is a “start-of-sequence” instruction identifying an iterative sequence of sub-optimal instructions that follow, and first, second, and third

operand identifiers 304, 306, and 308 suggest a number of loop iterations, a register to vary on each iteration, and a stride by which to increment the variable on each iteration. In an alternate embodiment, opcode 302 corresponds to a “start-of-sequence” instruction, and one or more of the first, second, and third operand identifiers 304, 306, and 308 are not used. In an alternate embodiment, opcode 302 corresponds to an “end-of-sequence” instructions, and first, second, and third operand identifiers 304, 306, and 308 are not used.

[0043] Table 1 illustrates an exemplary sequence of sub-optimal instructions optimized according to one embodiment. Specifically, the side-by-side-by-side code comparison of **Table 1** illustrates three versions of instructions: scalar code, AVX code, and Vectorbeam code, to implement the below loop, labeled as Code 1:

Code 1. *for (int i = 0; i < 64; i++) { outp[i] = inp1[i] + inp2[i]; }*

Scalar Code	
..LOOP	Label loop
movss (%rsp,%rax,4), %xmm0	Move 64 bits from (memory) to xmm0
movss 256(%rsp,%rax,4), %xmm1	Move 64 bits from (memory + 256) to xmm1
addss %xmm1, %xmm0	xmm0 += xmm1
movss %xmm0, 512(%rsp,%rax,4)	Move 64 bits from xmm0 to (memory + 512)
incq %rax	Increment rax
cmpq \$64, %rax	compare rax to 64
jle ..LOOP	Loop 64 Times
AVX Code	
..LOOP	Label loop
vmovups (%rsp,%rax,4), %ymm0	Vector move 256 bits from(memory) to ymm0

vmovups 256(%rsp,%rax,4), %ymm1	Vector move 256 bits from (memory + 256) to ymm1
vaddps %ymm1, %ymm0	Vector add ymm0 += ymm1
vmovups %ymm0,512(%rsp,%rax,4)	Vector move 256 bits from ymm0 to (memory + 512)
addq 8, %rax	Add 8 to rax
cmpq \$64, %rax	Compare rax to 64
jl ..LOOP	Loop 8 times
Vectorbeam Code	
foreach rax, 0, 64, 1	<i>Start-of-sequence:</i> loop on rax from 0 to 63 by 1
movss (%rsp,%rax,4), %xmm0	Move 64 bits from (memory) to xmm0
movss 256(%rsp,%rax,4), %xmm1	Move 64 bits from (memory + 512) to xmm1
addss %xmm1, %xmm0	xmm0 += xmm1
movss %xmm0, 512(%rsp,%rax,4)	Move 64 bits from xmm0 to (memory + 512)
next	

Table 1.

[0044] As shown in **Table 1**, scalar code includes a loop to be executed 64 times, incrementing register rax by one each time. Scalar code is sub-optimal because it may fail to exercise all parallel resources of the processor during its 64 loop iterations.

[0045] AVX code differs from scalar code in several respects. The AVX registers, ymm0 and ymm1 have different names and different sizes. The AVX opcodes have different names. Iteration through the data by 8 instead of by 1 is particular to the generation of processors associated with AVX. Generally, the work done to invent scalar instructions, such as the scalar instructions in **Table 1**, to document them, to enable compilers to emit them, to train users to use them, and to debug and collect traces of their execution may need to be partly or wholly repeated for reuse with each new generation of instruction set architecture (e.g. MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX 3.1, and AVX 3.2).

[0046] According to the disclosure, however, essentially the same scalar instructions with essentially the same format are used in Vectorbeam code. Specifically, the Vectorbeam code body is surrounded by a start-of-sequence instruction, here “foreach rax, 0, 64, 1,” and an end-of-sequence instruction, here “next.” As coded, the “foreach” opcode in the start-of-sequence instruction indicates to a processor implementing flow 100 (**Figure 1**) that the sub-optimal code to follow is meant to be iterative code. As coded, operands “rax” “0,” “64,” and “1” provide metadata suggesting to the processor implementing flow 100 (**Figure 1**) to loop the sub-optimal code using register rax as a loop index, varying rax from 0 to 63, with a stride of 1 for each iteration. In other words, the start-of-sequence instruction suggests to the processor implementing flow 100 (**Figure 1**) to execute the sub-optimal code 64 times, varying rax by 1 each time. The author of the original scalar code can prepare the Vectorbeam code sequence with essentially the same opcodes and essentially the same formats, and leave it to the processor to optimize the code. The processor will optimize the Vectorbeam code to take advantage of its particular hardware capabilities and instruction set architecture. In this way, the code sequence to implement Code 1 may be ported to new generations of processors and instruction set architectures without the effort of rewriting, redocumenting, reenabling, retesting, retraining, redebugging, and republishing the code. Developers may also spend less time learning new opcodes and new register files and other details associated with a new instruction set architecture when a new processor becomes available.

[0047] **Figure 4** illustrates an exemplary register renaming by a processor optimizing Vectorbeam code of Table 1 according to an embodiment. Specifically, Vectorbeam code in this embodiment is to be run by a 128-bit processor (e.g. SSE) that has a default strategy to expand Vectorbeam contexts by 4. Hence, the first time through the processor must perform code for rax having the values {0, 1, 2, 3}. When the processor reaches the Vectorbeam instruction, `movss (%rsp,%rax,4), %xmm0`, it must perform four loads:

$$xmm0[0] = (rsp + 0*4);$$

$$xmm0[1] = (rsp + 1*4);$$

$xmm0[2] = (rsp + 2 * 4);$

$xmm0[3] = (rsp + 3 * 4);$

[0048] Although Vectorbeam code of **Table 1** refers to architectural registers xmm0 and xmm1, the processor executing Vectorbeam code 400 of Figure 4 renames architectural registers to physical registers. (As is known to those of skill, the processor executing the code may have a large number of physical registers to serve as the architectural registers referenced in various program threads, threads that may be running simultaneously or out-of-order.) As shown, the processor accesses logical to physical register table 404 to determining how to map the architectural registers to physical registers. According to table 404, the processor assigns xmm0 and xmm1 to physical register file locations 4-7 and 12-15.

[0049] **Figure 5** illustrates an exemplary allotment of multiple computational resources used to process the allotted registers of Figure 4 in parallel according to one embodiment. The 128-bit SSE processor here applies its knowledge of its available hardware resources to optimize the code and the register renaming. As shown, the processor according to the disclosure renames the architectural xmm registers to the physical register file 502. The chosen register allotment in this embodiment allows four ALU's at a time to perform the addition in parallel, as indicated by 504, 506, 508, and 510 of **Figure 5**.

[0050] **Table 2** illustrate an exemplary sequence of Vectorbeam sub-optimal instructions being optimized according to one embodiment. Specifically, the side-by-side-by-side-by-side code comparison of **Table 2** illustrates four versions of instructions: scalar code, SSE code, AVX3 code, and Vectorbeam code, to implement the conditional loop of Code 2, below:

Code 2. *for (int i = 0; i < PTS; i++) { if (cond[i]) {outp[i] = inp1[i] + inp2[i];}}*

Scalar Code	SSE Code
..LOOP	..LOOP
cmpl \$0, 768(%rsp,%rax,4)	movups (%rsp,%rax,4), %xmm3

<pre> je ..FALSE movss (%rsp,%rax,4), %xmm0 addss 256(%rsp,%rax,4), %xmm0 movss %xmm0, 512(%rsp,%rax,4) ..FALSE incq %rax cmpq \$64, %rax jl ..LOOP </pre>	<pre> movdqu 768(%rsp,%rax,4), %xmm0 pcmpe4 %xmm1, %xmm0 addps 256(%rsp,%rax,4), %xmm3 movups 512(%rsp,%rax,4), %xmm4 pxor %xmm2, %xmm0 blendvps %xmm0, %xmm3, %xmm4 movups %xmm4, 512(%rsp,%rax,4) addq \$4, %rax cmpq \$64, %rax jb ..LOOP </pre>
AVX3 Code	Vectorbeam code
<pre> ..LOOP vmovups (%rsp,%rax), %zmm1 addb \$16, %dl vpcmpd \$4, 512(%rsp,%rax), %zmm0, %k1 vaddps 256(%rsp,%rax), %zmm1, %zmm2 vmovups %zmm2, 1624(%rsp,%rax){%k1} addq \$64, %rax cmpb \$64, %dl jb ..LOOP </pre>	<pre> foreach rax, 0, 64, 1 cmpl \$0, 768(%rsp,%rax,4) je ..FALSE movss (%rsp,%rax,4), %xmm0 addss 256(%rsp,%rax,4), %xmm0 movss %xmm0, 512(%rsp,%rax,4) ..FALSE Next </pre>

Table 2.

[0051] As shown, scalar code, like scalar code shown in **Table 1** and **Table 2**, is sub-optimal because it requires 64 loops. If scalar code or were to run on a modern vector processor, it would use less parallel hardware resources than are available.

[0052] As with AVX code (**Table 1**) a programmer migrating scalar code to SSE code would need to learn new opcodes, new register sets, and SSE code does not provide a jump bypass. Generally, the work done to invent the scalar instructions, document them, enable compilers to emit them, train users to use them, and to debug and collect traces for the code, which may need to be partly or wholly repeated each generation of instruction set architecture (e.g. MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX 3.1, and AVX 3.2).

[0053] SSE code, moreover, has a few shortcomings. For one, since the SSE code has no jump bypass, it performs the math for all the lanes regardless, and then uses a BLEND instruction to appropriately merge the untouched non-conditional values with those written in the conditional block. Performance suffers (a BLEND is ~1.5x slower than a normal store) - more registers are needed (5 vs. 2). As with AVX code (**Table 1**), SSE code generation requires a sophisticated compiler.

[0054] AVX3 code, like AVX code (**Table 1**), differs from scalar code in several respects. The AVX3 registers, ymm0...n have different names and different sizes. AVX3 opcodes have different names. AVX3 uses 'k' registers which are set as the result of a compare and then used in combination with stores / loads. Like SSE code, AVX3 code does not have a branch; the processor executes all opcodes under all conditions. Generally, then, the work done to invent the scalar instructions, document them, enable compilers to emit them, train users to use them, and to debug and collect traces for the code may need to be partly or wholly repeated for each generation of instruction set architecture (e.g. MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX 3.1, and AVX 3.2).

[0055] According to the disclosure, however, essentially the same scalar instructions as (**Table 1**) can be used with essentially the same format, in Vectorbeam code. Specifically, the Vectorbeam code starts with a start-of-sequence instruction, here "foreach rax, 0, 64, 1," and end-of-sequence instruction, here "next." As coded, the "foreach" opcode

indicates to a processor implementing flow 100 (**Figure 1**) that the sub-optimal Vectorbeam code to follow is iterative code. As coded, operands “rax” “0,” “64,” and “1” provide metadata suggesting to the processor implementing flow 100 (**Figure 1**) to loop the sub-optimal Vectorbeam code using register rax as a loop index, varying rax from 0 to 63, with a stride of 1 for each iteration. In other words, the processor implementing flow 100 (**Figure 1**) will be instructed to execute the sub-optimal code 64 times, varying rax by 1 each time. In this way, the code sequence to implement Code 1 may be ported to new generations of instruction set architectures without the effort of rewriting, retesting, and republishing the code. Developers may also spend less time learning new opcodes and new register files and other details associated with a new instruction set architecture.

[0056] Moreover, according to the disclosure, the CPU architect, who has knowledge of the particular resources, capabilities, and instruction set architecture associated with a particular CPU can choose how to implement the functions described in the sequence of sub-optimal Vectorbeam instructions. Furthermore, the opcode 302 (Figure 3) and the metadata 304, 306, and 308 (Figure 3) can provide hints to the processor in choosing what instructions to substitute for the sequence of sub-optimal Vectorbeam instructions. For example, the start-of-sequence instruction can let the processor know that a number of memory loads or stores utilize the same offset. Or, the start-of-sequence instruction can provide a hint that a number of memory loads or stores are to the same cache line. Or, the start-of-sequence instruction can provide a hint that a number of mathematical operations use the same operand, as might be the case in a multimedia or graphics code routine that adjusts pixels uniformly, such as by increasing brightness.

[0057] **Figure 6** illustrates a portion of a lookup table listing vector instruction substitutes for scalar instructions according to one embodiment. Specifically, table 600 lists scalar instructions and their corresponding vector equivalents for performing mathematical operations: addition, subtraction, multiplication, division, square root, maximum, minimum, reciprocal, and reciprocal of square root. Referring to side-by-side table 600, in one embodiment, the sequence of sub-optimal Vectorbeam instructions stored in instruction buffer 212 (**Figure 2**) includes a scalar addition, ADDSS, and substitute

evaluation logic 216 (**Figure 2**) substitutes a vector addition, ADDPS, for one or more of the sub-optimal Vectorbeam instructions.

[0058] **Figure 7** illustrates a vector processor register file according to one embodiment. Specifically, register file 700 includes vector registers 702 and general purpose registers 704. As shown, vector registers 702 include 32 zmm registers 706, each 512 bits wide, 16 ymm registers 708, each 256 bits wide, and 16 xmm registers 710, each 128 bits wide. As part of evaluating substitutes for sub-optimal Vectorbeam instructions in instruction buffer 212 (**Figure 2**), substitute evaluation logic 216 (**Figure 2**) in one embodiment assigns the results of 8 32-bit scalar operations to a 256-bit ymm register.

[0059] In an alternate embodiment, a Vectorbeam sequence of sub-optimal instructions is replaced with instructions from a newer instruction set architecture. In one embodiment a sequence of sub-optimal SSE instructions that utilize 128-bit XMM registers 704 is replaced by a sequence of AVX, running instructions that utilize 256-bit YMM registers 706 and achieve better performance without requiring the sub-optimal code to be rewritten by its author. In another embodiment, a sequence of sub-optimal AVX instructions that utilize YMM registers 706 are replaced by a sequence of AVX-512 instructions that utilize ZMM registers 708 and achieve better performance without requiring the sub-optimal code to be rewritten by its author.

[0060] In an alternate embodiment, opcode 302 (**Figure 3**) could identify the particular instruction set architecture associated with the sub-optimal instructions stored in instruction buffer 212 (**Figure 2**). Substitute evaluation logic 216 (**Figure 2**) could then substitute optimal instructions associated with the processor's instruction set architecture for the sub-optimal instructions.

[0061] In one embodiment, the sub-optimal instructions and the substitute instructions are both vector instructions, but the substitute instructions are from a newer generation and use wider registers.

CLAIMS

What is claimed is:

1. A processor, comprising:

a fetch circuit to fetch an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction;

a decode circuit to decode the instruction; and

an execution circuit to execute a dispatched instruction received from the decode circuit;

wherein the decode circuit is configured to detect the start-of-sequence instruction and the end-of-sequence instruction, to buffer the sequence of sub-optimal instructions there between, to access a lookup table to identify a sequence of one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and to select either the decoded instruction or the sequence of one or more optimized instructions to dispatch to the execution circuit.

2. A processor, comprising:

a fetch circuit to fetch an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction;

a decode circuit to decode the instruction, the decode circuit comprising means for detecting the start-of-sequence instruction and the end-of-sequence instruction, and means for buffering the sequence of sub-optimal instructions there between; and

an execution circuit to execute a dispatched instruction received from the decode circuit;

wherein the decode further to access a lookup table to identify a sequence of one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and to select either the decoded instruction or the sequence of one or more optimized instructions to dispatch to the execution circuit.

3. Any of the processors of claims 1-2, wherein the sequence of sub-optimal instructions comprises scalar instructions and the sequence of one or more optimized instructions comprise vector instructions.
4. Any of the processors of claims 1-2, wherein the sequence of sub-optimal instructions comprises scalar instructions and the sequence of one or more optimized instructions comprise Single Instruction Multiple Data instructions.
5. Any of the processors of claims 1-4, wherein the metadata describes an instruction set architecture associated with the sub-optimal instructions, and wherein the sequence of one or more optimized instructions are associated with another instruction set architecture.
6. Any of the processors of claims 1-5, wherein the sequence of one or more optimized instructions utilize wider registers than do the sequence of sub-optimal instructions.
7. Any of the processors of claims 1-6, wherein the decode circuit is further configured to access the lookup table to identify a plurality of substitute instructions to substitute for a plurality of the sequence of sub-optimal instructions.
8. Any of the processors of claims 1-7, wherein the first source operand identifier and the second source operand identifier of the start-of-sequence instruction comprise metadata that provides data describing the sequence of sub-optimal instructions.
9. Any of the processors of claims 1-8, wherein the sequence of sub-optimal instructions loop over an iterative sequence of execution.
10. Any of the processors of claims 1-9, wherein the first source operand identifier specifies a number of times to iterate over the sequence of sub-optimal instructions, the second source operand identifier specifies a variable to increment during each iteration, and a third source operand identifier specifies a stride by which to increment the variable during each iteration.

11. A method, comprising:
- fetching an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction;
 - decoding the instruction; and
 - executing a dispatched instruction received from the decode circuit;
- wherein the decoding further comprises detecting the start-of-sequence instruction and the end-of-sequence instruction, buffering the sequence of sub-optimal instructions there between, accessing a lookup table to identify a sequence of one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and selecting either the decoded instruction or the sequence of one or more optimized instructions to execute.
12. A method, comprising:
- fetching an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction;
 - decoding the instruction, including performing steps for detecting the start-of-sequence instruction and the end-of-sequence instruction, and steps for buffering the sequence of sub-optimal instructions there between; and
 - executing a dispatched instruction received from the decode circuit;
- wherein the decoding further comprising accessing a lookup table to identify a sequence of one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and selecting either the decoded instruction or the sequence of one or more optimized instructions to execute.
13. Any of the methods of claims 11-12, wherein the sequence of sub-optimal instructions comprises scalar instructions and the sequence of one or more optimized instructions comprise vector instructions.

14. Any of the methods of claims 11-12, wherein the sequence of sub-optimal instructions comprises scalar instructions and the sequence of one or more optimized instructions comprise Single Instruction Single Instruction Multiple Data instructions.
15. Any of the methods of claims 11-14, wherein the metadata describes an instruction set architecture associated with the sub-optimal instructions, and wherein the sequence of one or more optimized instructions are associated with another instruction set architecture.
16. Any of the methods of claims 11-15, wherein the sequence of one or more optimized instructions utilize wider registers than do the sequence of sub-optimal instructions.
17. Any of the methods of claims 11-16, wherein the decode circuit further to access the lookup table to identify a plurality of substitute instructions to substitute for a plurality of the sequence of sub-optimal instructions.
18. Any of the methods of claims 11-17, wherein the first source operand identifier and the second source operand identifier of the start-of-sequence instruction comprise metadata that provides data describing the sequence of sub-optimal instructions.
19. Any of the methods of claims 11-18, wherein the sequence of sub-optimal instructions loop over an iterative sequence of execution.
20. Any of the methods of claims 11-19, wherein the first source operand identifier specifies a number of times to iterate over the sequence of sub-optimal instructions, the second source operand identifier specifies a variable to increment during each iteration, and a third source operand identifier specifies a stride by which to increment the variable during each iteration.

21. An article of manufacture comprising a non-transitory machine-readable storage medium, the non-transitory machine-readable storage medium storing instructions capable of being executed by a processor to perform a procedure comprising:

fetching an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction;

decoding the instruction; and

executing a dispatched instruction received from the decode circuit;

wherein the decoding further comprises detecting the start-of-sequence instruction and the end-of-sequence instruction, buffering the sequence of sub-optimal instructions there between, accessing a lookup table to identify a sequence of one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and selecting either the decoded instruction or the sequence of one or more optimized instructions to execute.

22. An article of manufacture comprising a non-transitory machine-readable storage medium, the non-transitory machine-readable storage medium storing instructions capable of being executed by a processor to perform a procedure comprising:

fetching an instruction from an instruction storage, a format of the instruction including an opcode, a first source operand identifier, and a second source operand identifier; wherein the instruction storage includes a sequence of sub-optimal instructions preceded by a start-of-sequence instruction and followed by an end-of-sequence instruction;

decoding the instruction, including performing steps for detecting the start-of-sequence instruction and the end-of-sequence instruction, and steps for buffering the sequence of sub-optimal instructions there between; and

executing a dispatched instruction received from the decode circuit;

wherein the decoding further comprising accessing a lookup table to identify a sequence of one or more optimized instructions to substitute for one or more of the sequence of sub-optimal instructions, and selecting either the decoded instruction or the sequence of one or more optimized instructions to execute.

23. Any of the articles of manufacture of claims 22-23, wherein the sequence of sub-optimal instructions comprises scalar instructions and the sequence of one or more optimized instructions comprise vector instructions.

24. Any of the articles of manufacture of claims 22-23, wherein the sequence of sub-optimal instructions comprises scalar instructions and the sequence of one or more optimized instructions comprise Single Instruction Multiple Data instructions.

25. Any of the articles of manufacture of claims 22-25, wherein the metadata describes an instruction set architecture associated with the sub-optimal instructions, and wherein the sequence of one or more optimized instructions are associated with another instruction set architecture.

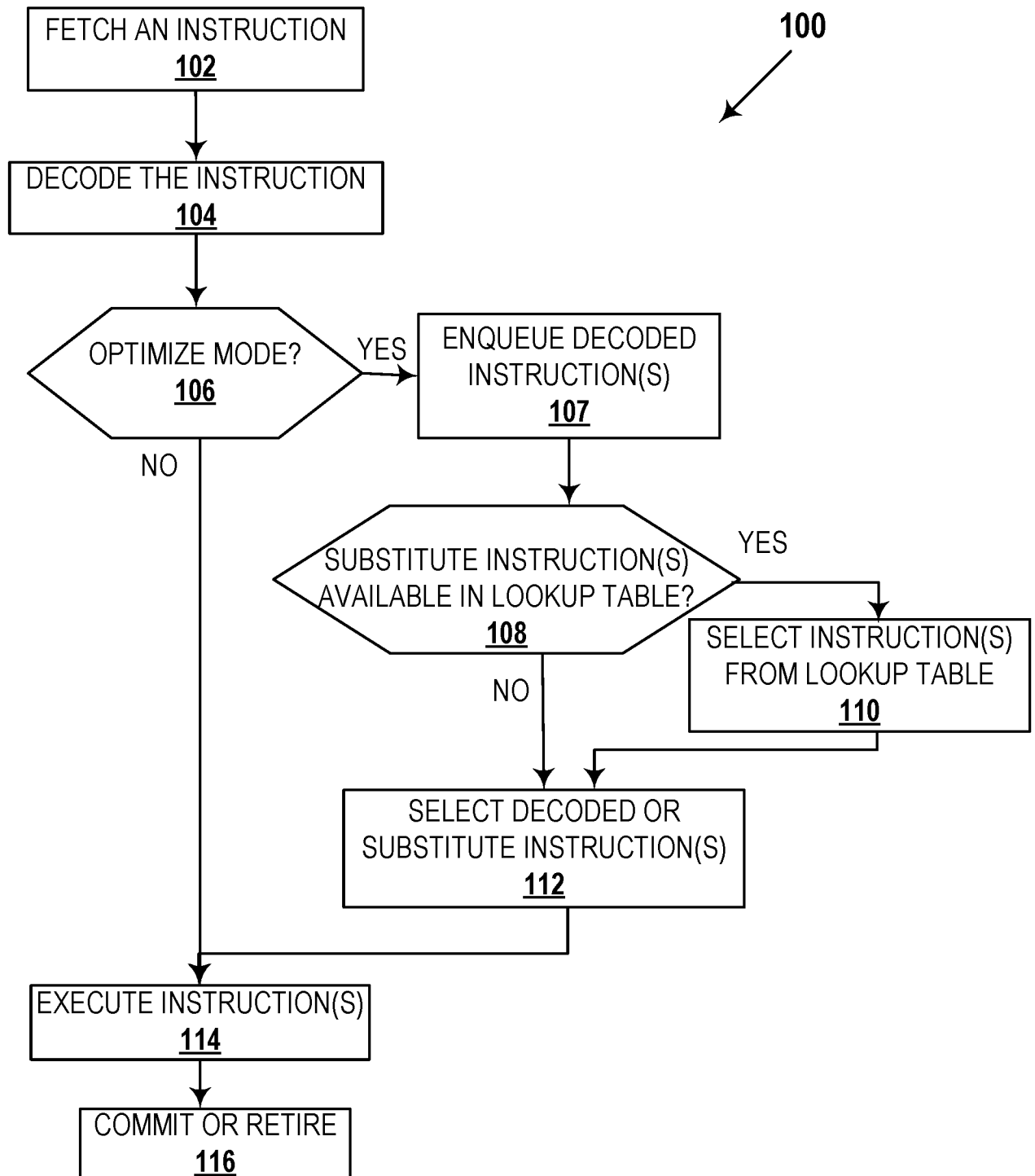


FIG. 1

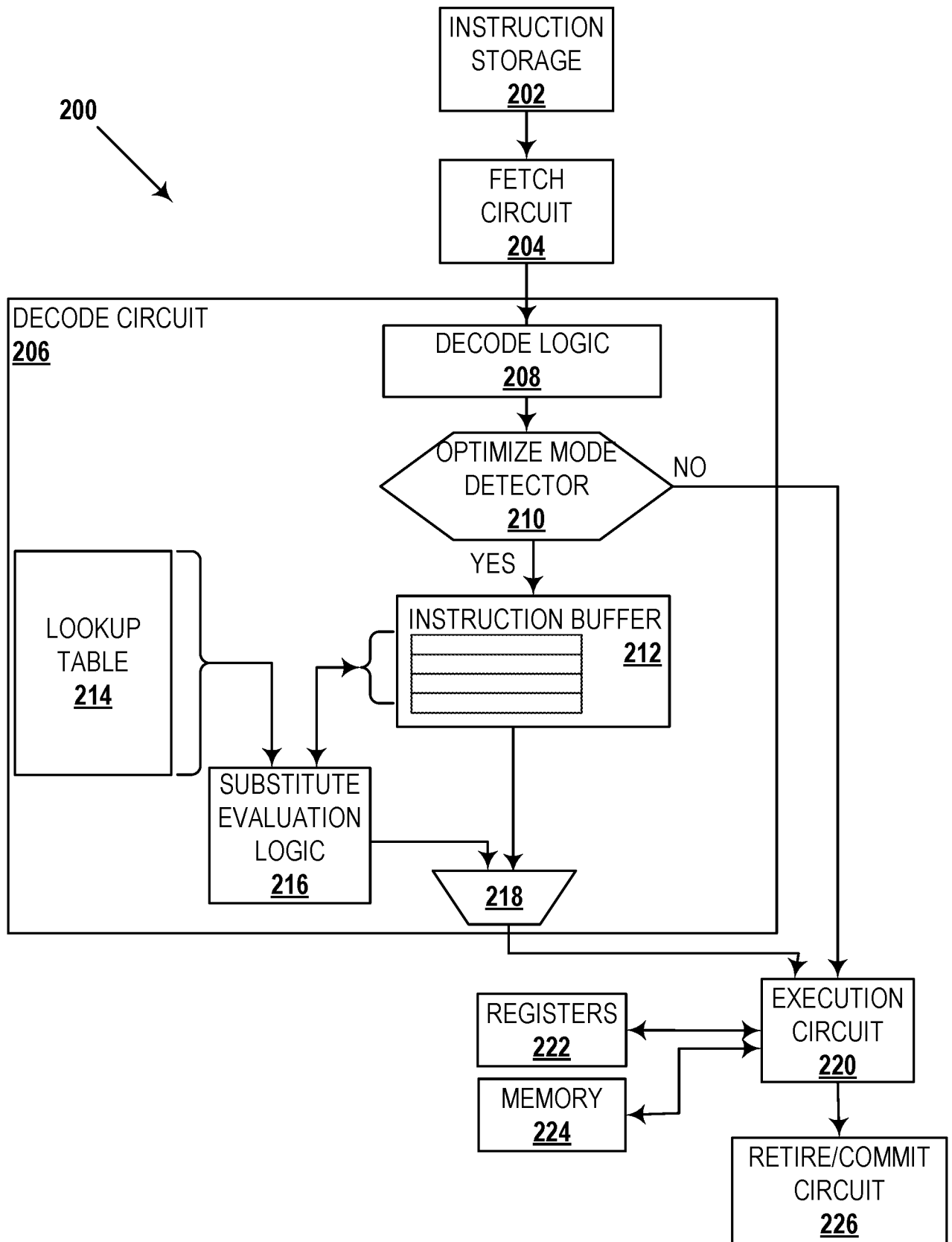


FIG. 2

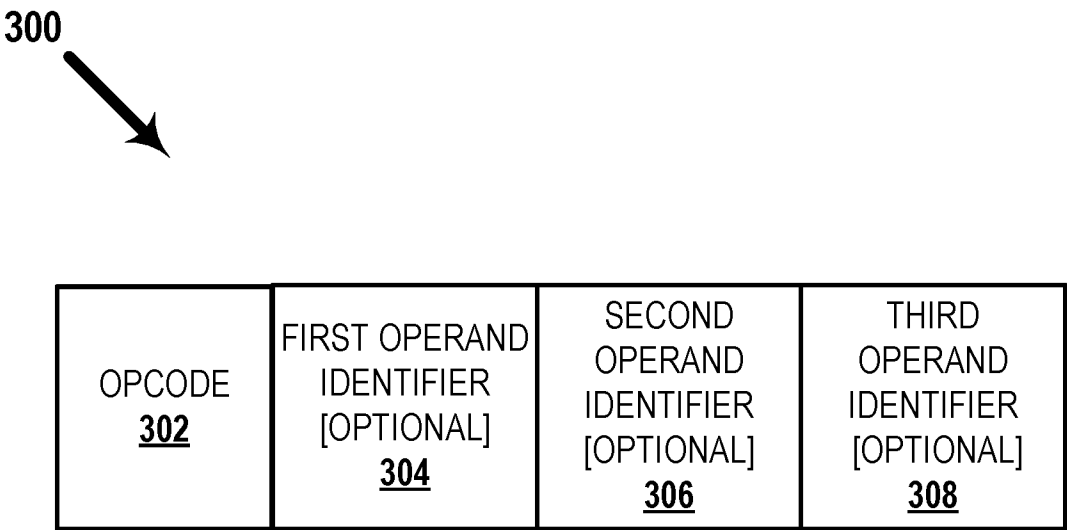


FIG. 3

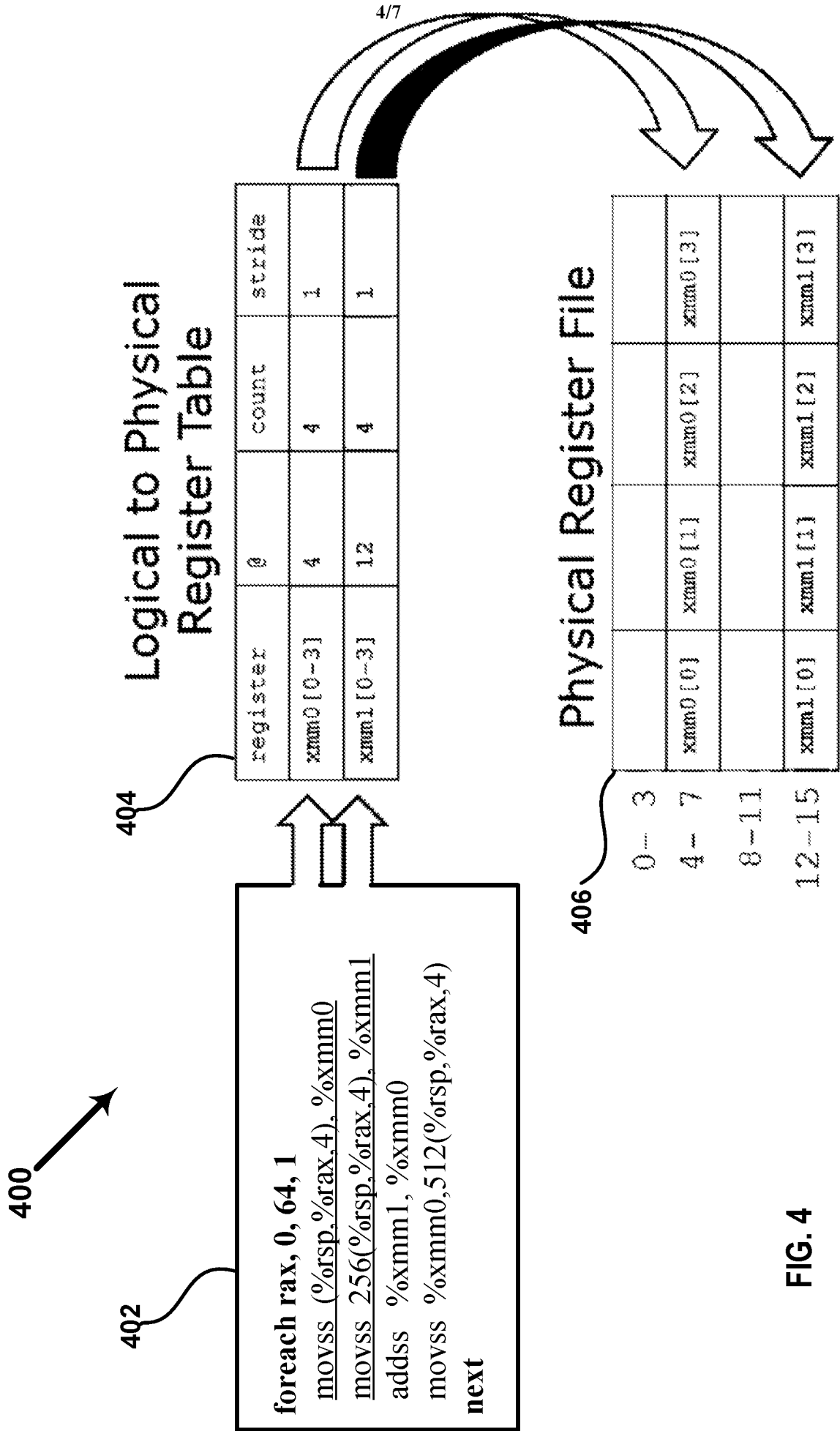


FIG. 4

500

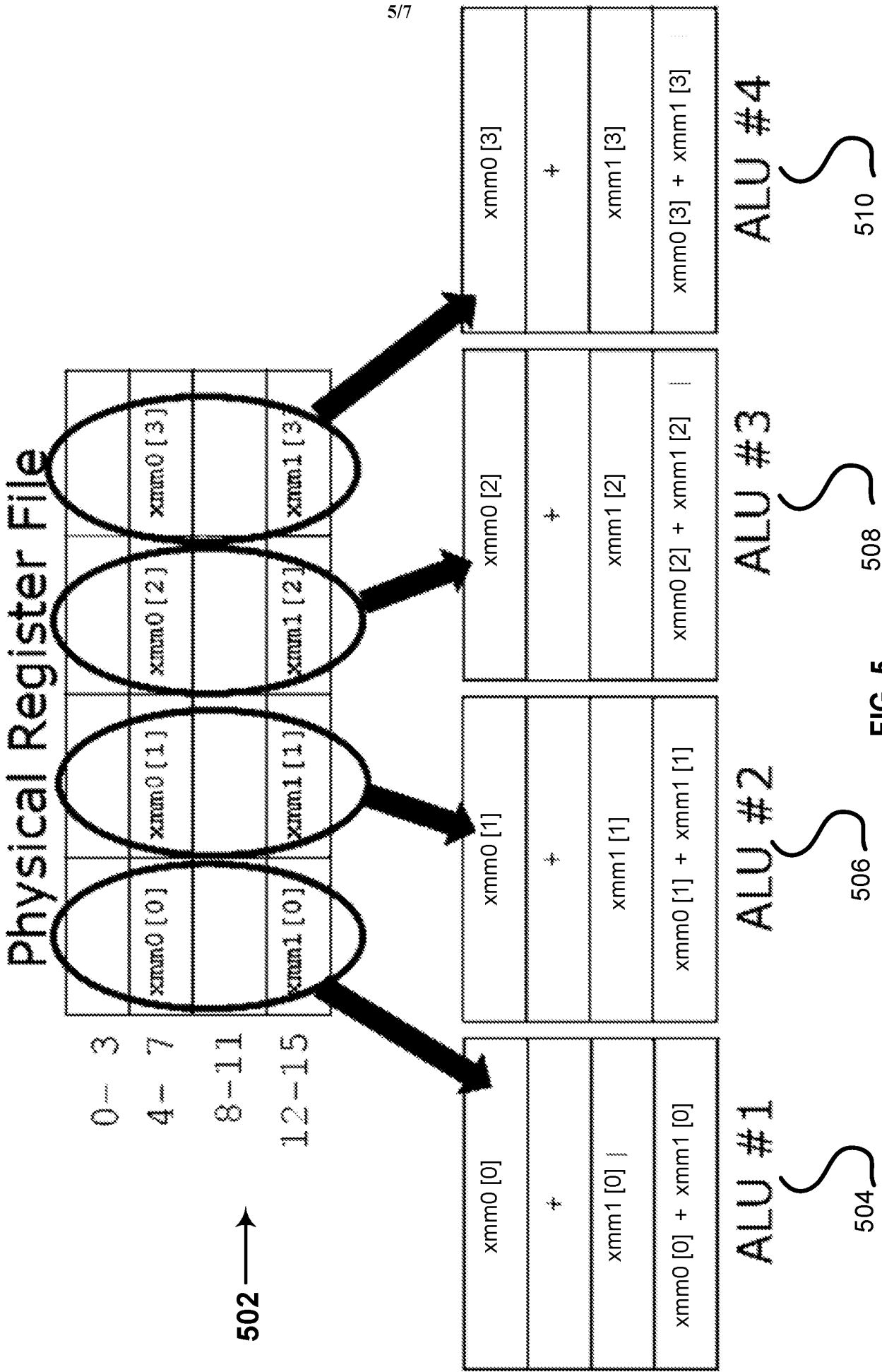


FIG. 5

600

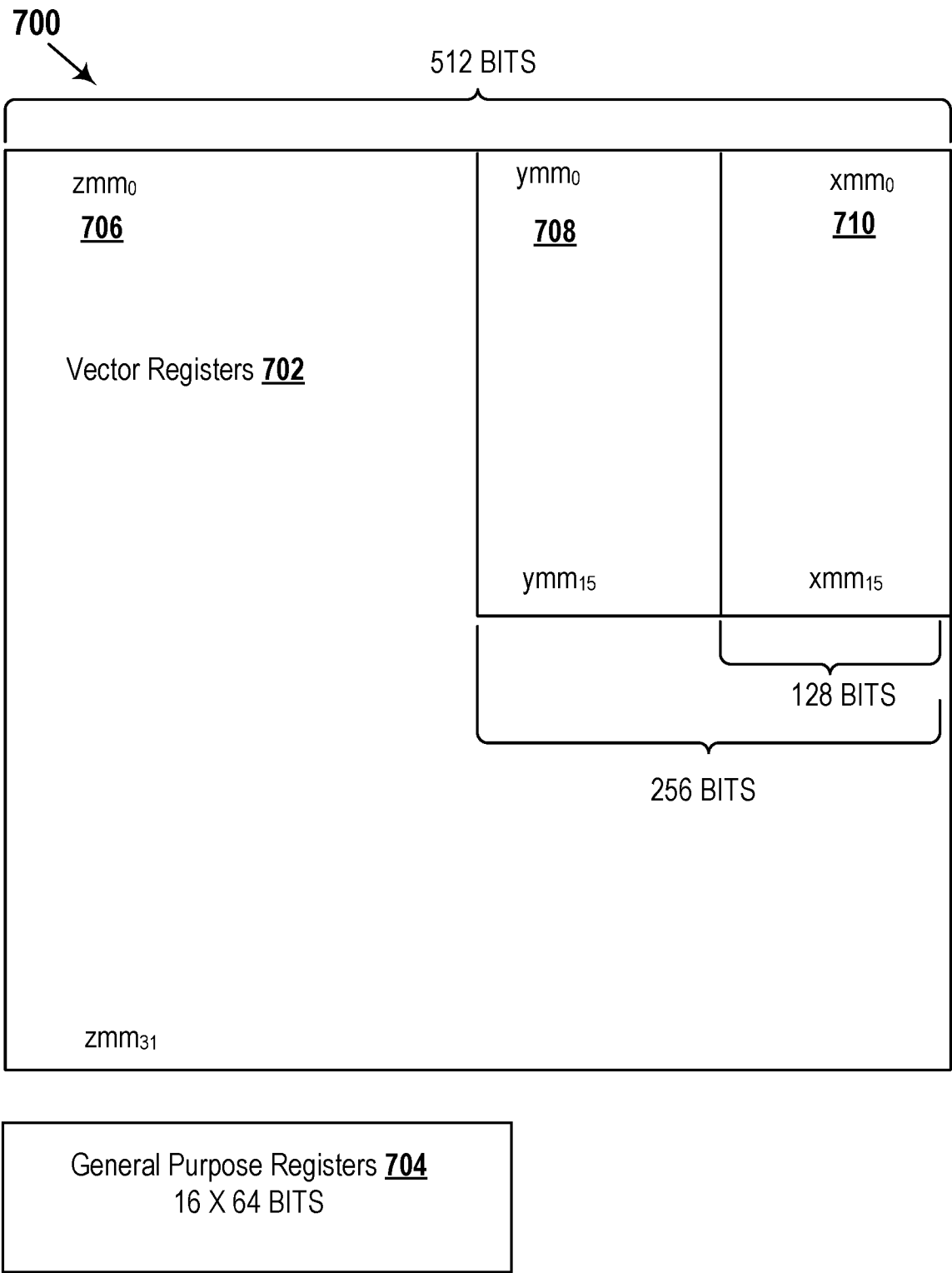


SCALAR VS. VECTOR MATHEMATICAL INSTRUCTIONS

Scalar□	Vector□	Description□
ADDSS□	ADDPS□	Adds operands□
SUBSS□	SUBPS□	Subtracts operands□
MULSS□	MULPS□	Multiplies operands□
DIVSS□	DIVPS□	Divides operands□
SQRTSS□	SQRTPS□	Square root of operands□
MAXSS□	MAXPS□	Maximum of operands□
MINSS□	MINPS□	Minimum of operands□
RCPSS□	RCPPS□	Reciprocal of operands□
RSQRTSS□	RSQRTPS□	Reciprocal of square root of operands□

FIG. 6

FIG. 7



A. CLASSIFICATION OF SUBJECT MATTER**G06F 9/30(2006.01)i, G06F 9/48(2006.01)i**

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 9/30; G06F 9/44; G06F 9/06; G06F 15/76; G06F 9/45; G06F 9/48

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKOMPASS(KIPO internal) & Keywords: SIMD, instruction optimization, sub-optimal, scalar, vector, substitute, start, end, delimiter**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2013-0067196 A1 (GERALD PAUL MICHALAK et al.) 14 March 2013 See paragraphs [0040]-[0043], [0046]-[0047], [0051]-[0052], [0055]; and figures 1-3.	1-7, 9, 11-17, 19, 21-25
Y		8, 10, 18, 20
Y	US 2014-0189331 A1 (MARIA LIPSHITS et al.) 03 July 2014 See paragraphs [0068], [0070]; and figure 5.	8, 10, 18, 20
A	US 2014-0237460 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 21 August 2014 See paragraphs [0059]-[0061]; and figures 8-9.	1-25
A	US 2013-0290943 A1 (TAL ULIEL et al.) 31 October 2013 See paragraphs [0018], [0021]-[0024]; and figures 1-2.	1-25
A	US 2014-0096119 A1 (NALINI VASUDEVAN et al.) 03 April 2014 See paragraphs [0031]-[0040]; and figures 3-4B.	1-25



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

26 June 2017 (26.06.2017)

Date of mailing of the international search report

26 June 2017 (26.06.2017)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea



Facsimile No. +82-42-481-8578

Authorized officer

CHIN, Sang Bum

Telephone No. +82-42-481-8398



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/015052

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2013-0067196 A1	14/03/2013	WO 2013-040271 A1	21/03/2013
US 2014-0189331 A1	03/07/2014	US 9459871 B2	04/10/2016
US 2014-0237460 A1	21/08/2014	US 2014-0237217 A1	21/08/2014
		US 9047077 B2	02/06/2015
		US 9052888 B2	09/06/2015
US 2013-0290943 A1	31/10/2013	CN 103999045 A	20/08/2014
		TW 201344575 A	01/11/2013
		TW I502506 B	01/10/2015
		US 2015-0154008 A1	04/06/2015
		US 8984499 B2	17/03/2015
		WO 2013-089750 A1	20/06/2013
US 2014-0096119 A1	03/04/2014	US 2016-0139897 A1	19/05/2016
		US 9244677 B2	26/01/2016